

Information Retrieval on the Connection Machine: 1 to 8192 Gigabytes

Craig Stanfill

Robert Thau

Thinking Machines Corporation

245 First Street

Cambridge MA 02154

ABSTRACT

DR90-3

This paper describes algorithms and data structures for applying a parallel computer to information retrieval. Previous work has described an implementation based on overlap encoded signatures. That system was limited by 1) the necessity of keeping the signatures in primary memory, and 2) the difficulties involved in implementing document-term weighting. Overcoming these limitations requires adapting the inverted index techniques used on serial machines. The most obvious adaptation, also previously described, suffers from the fact that data must be sent between processors at query-time. Since interprocessor communication is generally slower than local computation, this suggests that an algorithm which does not perform such communication might be faster. This paper presents a data structure, called a partitioned posting file, in which the interprocessor communication takes place at database-construction time, so that no data movement is needed at query-time. Performance characteristics and storage overhead are established by benchmarking against a synthetic database. Based on these figures, it appears that currently available hardware can deliver interactive document ranking on databases containing between 1 and 8192 Gigabytes of text.

This is a pre-print of a paper to appear on *Information Processing and Management* in 1991. It should not be redistributed. Copyright 1991 *Information Processing and Management*.

Information Retrieval on the Connection Machine: 1 to 8192 Gigabytes

*Craig Stanfill
Robert Thau
Thinking Machines Corporation
245 First Street
Cambridge MA 02154*

ABSTRACT

This paper describes algorithms and data structures for applying a parallel computer to information retrieval. Previous work has described an implementation based on overlap encoded signatures. That system was limited by 1) the necessity of keeping the signatures in primary memory, and 2) the difficulties involved in implementing document-term weighting. Overcoming these limitations requires adapting the inverted index techniques used on serial machines. The most obvious adaptation, also previously described, suffers from the fact that data must be sent between processors at query-time. Since interprocessor communication is generally slower than local computation, this suggests that an algorithm which does not perform such communication might be faster. This paper presents a data structure, called a partitioned posting file, in which the interprocessor communication takes place at database-construction time, so that no data movement is needed at query-time. Performance characteristics and storage overhead are established by benchmarking against a synthetic database. Based on these figures, it appears that currently available hardware can deliver interactive document ranking on databases containing between 1 and 8192 Gigabytes of text.

1. Introduction

Immense quantities of data are currently being captured in electronic form. Most documents are composed using word processing or desktop publishing systems. Much correspondence takes place in electronic form. Electronic bulletin boards and mailing lists provide for the fast dissemination of information. Increasing numbers of newspapers and magazines are available in electronic form. The scientific community is gradually moving towards electronic publishing. This growing body of electronic text is a potentially important resource, but only if it is possible to locate desired information in a timely manner.

For relatively small quantities of electronic information, the methods of library science — cataloging, assigning keywords, and careful organization — are usable. When coupled with text database systems based on boolean keyword searches, they provide a method which, in the hands of a skilled searcher, permits at least some information to be located. Questions as to the effectiveness of these methods do remain, however, and it is unclear whether, even in the hands of an expert, these systems provide sufficiently high-quality searches. Non-experts generally find these systems difficult to use effectively. In any event, the librarianship required to build these electronic repositories is labor intensive, and the vast majority of electronic text is completely unorganized.

There is at least a partial solution to this problem, in the form of automatic information retrieval on full-text databases¹. In a typical system of this class, such as Salton's SMART system (1971), the full text will be passed through an automatic indexing system which will reduce a document to a set of *terms* (which might be words, word-stems, or keywords) and *weights* representing the importance of those terms to the content of the document. When a user queries the database, he will enter a set of *query terms*. A second automatic method will assign weights to the query terms, and a *document scoring and ranking* algorithm will be employed to find those documents most exactly matching the user's query; this is the well known vector retrieval model, as described by Salton (1975). The user may then browse these documents and, if necessary, refine the query either manually or via an automatic method, for example relevance feedback as used by Rocchio (1972); the documents in the database will then be re-ranked, and the process repeats until the user believes he has found a sufficient number of documents. This method has been found to deliver a high-quality search and, when combined with a graphic interface, is quite easy to use.

One key component of such a system is the document scoring and ranking algorithm. In a conventional inverted file algorithm, as described by Salton (1989), the output of the indexer will be sorted by term identifier, then the result stored on disk. When the user enters a query, those index entries needed to answer the query are loaded into memory and any of several algorithms is employed to score and rank the documents. This file structure has the advantage of minimizing the amount of data to be transferred, and requires a modest number of I/O operations. For small databases, the performance of the system will be limited by disk latency. As the database grows larger, the disk-to-memory transfer time and the time needed to score and rank the documents becomes increasingly important, and eventually a point will be reached where the system is too slow to be considered interactive. The I/O throughput can be improved by using a mainframe computer with a large number of disks, but as long as a serial machine is used, there are limited prospects for improving the performance of the scoring and ranking algorithms.

Parallel computers, with higher basic compute rates coupled with very high disk-to-memory bandwidths, are a promising alternative to serial machines for the solution of this problem. Systems based on parallel adaptations of the serial inverted file structure are particularly promising in this respect. This paper will present an inverted file structure coupled with algorithms for document scoring and document ranking which, on the Connection Machine[®] System, deliver over 200 times the performance attainable on a state-of-the-art serial machine (Sun-4/330). Depending on the hardware configuration and the I/O strategy employed, several different system architectures may be implemented. One implementation, storing the database entirely in primary memory, can handle databases with up to 24 Gigabytes of data, delivering response times under 100 milliseconds. A second implementation, using a disk-array optimized for large numbers of I/O's per second, is suitable for databases with between 3 and 128 Gigabytes of data, and delivers responses in between 0.6 and 2 seconds. A third implementation, using a disk array optimized for high transfer rates, is suitable for databases with between 128 and 8192 Gigabytes of data, and delivers responses in between 2 and 15 seconds.

1. See, for example, vanRijsbergen (1979) and Salton (1989).

[®] Connection Machine is a registered trademark of Thinking Machines Corporation.

1.1 Previous Work

Initial parallel implementations of information retrieval by Stanfill and Kahle (1986), and by Pogue and Willet (1987), were based on overlap-encoded signatures. Overlap encoding is subject to some very specific constraints:

- According to Stone (1987), if the signature file does not fit in primary memory the I/O load will prevent interactive access.
- Only binary document weights can be supported; according to Salton and Buckley (1988) and Croft (1988) this may reduce the quality of the search, relative to systems which support full vector weighting.

Within the above constraints, Stanfill (1988, 1990a) argues that signature files deliver excellent response times. However, these constraints somewhat limit the usefulness of the method.

Stone (1987) has suggested that parallel inverted files might not be subject to the above limitations. Stanfill, Thau, and Waltz (1989) have described one possible parallel inverted file structure for use on the Connection Machine. Stanfill (1990b) has described a second structure called partitioned posting files, which alters the data layout so as to reduce interprocessor communication. This paper expands on the previously published description of the partitioned posting file structure.

1.2 Paper Organization

This paper will describe the application of parallel partitioned files to databases between 1 and 8192 Gigabytes, on massively parallel computers with between 4096 and 65,536 processors. Section 2 presents some background in information retrieval and parallel computing. Section 3 describes serial as well as parallel inverted file schemes. Section 4 describes the partitioned posting file structure and presents some results on the efficiency and storage overhead involved in this structure. Section 5 describes how partitioned posting files may be used to score documents. Section 6 describes how documents, once scored, may be ranked. Section 7 describes two I/O strategies for use with partitioned posting files. Section 8 describes three different system architectures which are suitable for differing sizes of databases. Section 9 summarizes the results and discusses directions for further research.

1.3 A Cautionary Note

The performance results presented below are based on benchmarks against a synthetic database. Caution must be used in interpreting the results of such studies. First, no real database is going to exactly match the synthetic one and, as a result, the performance of the system on real data is likely to be different the results presented here. Second, benchmark figures are not the same as application-level performance. Benchmarks do not account for system overhead, queuing delays, and the stages of processing which come before or after the actual retrieval step. They do not include the sort of exception-handling and error-detection needed to build a robust application. They do not account for the storage of the actual full text. However, the synthetic database does model a realistic workload, and the document scoring/ranking step does constitute the bulk of the work required to search a database. In any event, the benchmark figures do reflect the relative performance of various methods of solving this problem.

2. Background

2.1 Information Retrieval

A document is represented as an unordered set of weighted terms. The weights indicate the importance of each term within the document. The terms may correspond to words, or to word-stems, or to phrases, or to high-level concepts. The extraction of terms and assignment of weights is generally done automatically, as suggested by Salton (1970). Queries also consist of weighted terms, and are generally produced by a combination of automatic and manual methods; such methods are discussed in detail by Salton (1987).

The operation of a retrieval system can best be described in terms of the *vector model*. A database consists of a set of documents and a vocabulary of n terms T_i . A document D is represented as a vector of length n such that $D_i > 0$ only if T_i is present in D . A query Q has the same representation. Retrieval is based on some measurement of document-query similarity. This paper will assume the *cosine similarity measure* (Equation 1).

$$\text{similarity}(D, Q) = \frac{D \cdot Q}{\|D\| \|Q\|} \quad (1)$$

The basic computation of information retrieval is finding the document D which maximizes $\text{similarity}(D, Q)$.

Retrieval methods based on this vector model differ significantly from those provided by most commercially available text databases (e.g. Westlaw, Lexis, Dialog). The most obvious difference is that the commercial systems employ boolean connectives such as AND/OR rather than numerical weights. A less obvious distinction has to do with indexing. The vector model, as formulated above, assumes that a document is automatically indexed so as to reduce it to a set of terms which reflect its content, with weights reflecting the importance of those terms. If "terms" are equated with "words" or "word stems," difficulties will inevitably arise. For example, if a document containing the text-string "New Mexico" is indexed as containing only the terms "New" and "Mexico," then that document may be effectively un retrievable: for every document containing "New Mexico" there may be a hundred which coincidentally contain "New" and "Mexico," but refer to the country rather than the state. The commercial systems mentioned above overcome this problem by recording the location of every word in the database and providing a proximity operator which allows the user to search for "New" immediately followed by "Mexico." These proximity operators are extremely powerful, and allow for a number of retrieval strategies which depend on the context in which a term occurs.

Similar difficulties arise from the morphological structure of language. English morphology may, for the most part, be handled at indexing-time by stripping suffixes from words to arrive at word-stems, or at query-time by allowing the user to incorporate wild-cards at the end of search terms (tail truncation). Such simple strategies may not suffice in other languages. For example, German uses long compound nouns; searching a German database might thus require both left- and right-wildcards in search terms, or extensive morphological analysis to break the compounds into their component nouns. In Japanese, the boundaries between words are not indicated by spaces, resulting in considerable ambiguity. Searching a Japanese database might thus require either arbitrary substring search, or some form of linguistic analysis to indicate which words are actually present.

Effective use of the algorithms as described below thus depends on proper automatic indexing, which must include as a minimum some mechanism for detecting multi-word proper nouns such as New Mexico. In cases where such indexing is not possible, proximity-based methods might be mandatory. The algorithms and data structures described below can probably be modified to incorporate proximity operators, but a discussion of such issues is beyond the scope of this paper. Ultimately, however, improved automatic indexing is probably more desirable than simply making complex proximity-based search methods run faster.

2.2 A Synthetic Database

Evaluation of scoring and ranking algorithms will be done using a synthetic database and query load described by Stanfill (1989). This has the advantage that 1) the database may be easily replicated by other researchers; 2) the parameters of the database (such as size) may be altered to explore the behavior of retrieval algorithms; 3) large quantities of disk need not be tied up; and 4) generation of the database can be selective: given a 10-term query, only that portion of the posting file needed to evaluate that query needs to be synthesized.

The lexicon for the database consists of n terms, T_1 through T_n . The frequency $f(T_i)$ of T_i (occurrences per megabyte) is given by equation 2, in accordance with Zipf's law. Terms 1 through s are *stop words*, and are not put into the posting file. It is assumed that the terms found in queries have the same frequency distribution as the terms in the database, omitting stop-words. The probability distribution of a random query term Q_j is given by equation 3. Equation 3 is, of course, subject to the constraint that Q_j be a probability distribution (Equation 4).

$$f(T_i) = \frac{c_1}{i} \quad (2)$$

$$\Pr(Q_j = T_i) = \begin{cases} i < s & 0 \\ \text{otherwise} & \frac{c_2}{i} \end{cases} \quad (3)$$

$$1 = \sum_{i=s}^n \frac{c_2}{i} \quad (4)$$

The amount of work done by an inverted file algorithm is governed by how many times a query-term occurs in the database. For example, a rare term might occur only once, and hence involve a relatively small amount of work; a common term might occur a million times, and hence involve a larger amount of work. The distribution of query-term frequencies is thus of great importance. This distribution is $f(Q_j)$, and the expected frequency of a randomly selected query-term is $Z = E(f(Q_j))$. (Equation 5).

$$Z = E(f(Q_j)) = \sum_{i=s}^n f(T_i) \Pr(Q_j = T_i) = \sum_{i=s}^n \frac{c_1 c_2}{i^2} \quad (5)$$

The amount of disk required to store an inverted file is determined by the total number of non-stop words in the database which depends, in turn on the total term-frequency R_T (Equation 6). The total number of terms in a database of $|D|$ megabytes will then be $R_T|D|$.

$$R_T = \sum_{i=1}^{i=n} f(T_i) = \sum_{i=1}^{i=n} \frac{c_1}{i} \quad (6)$$

In practice, it is not possible to directly measure c_1 or c_2 . One may, however, measure Z and R_T . For a database of newswire articles a value of $Z = 3$ is reasonable (i.e. a randomly selected query-term occurs three times per megabyte). Similarly, a value of $R_T = 58,000$ (i.e. one term per 17.3 bytes) is reasonable for newswire data. At this point one may freely choose one of the four parameters (in this case, $n = 200,000$), and use the constraints in equations 4, 5, and 6 to determine values for s , c_1 , and c_2 . These values are given in Table 1.

n	200,000
s	550
c_1	9778
c_2	.1696

Table 1: Database Parameters for Newspaper Articles

Finally, the performance of the ranking algorithm depends on the number of documents in the database and the number of documents to be actually ranked and returned to the user. This paper will assume an average document-length of 5000 bytes, and a rank-count of 20. The query length (number of terms per query) is also important in predicting system performance. This system will assume two query-types might be desired: relatively small queries having 10 terms, and somewhat larger queries with 30 terms. The former might be generated manually; the later might be created by a semi-automatic method such as thesaurus-based expansion or relevance feedback.

Six parameters — database size, average query-term frequency, query length, rate of occurrence of non-stop-words, average document size and number of documents returned — will be different from one database to another, but have essentially independent effects. The database size, average query-term frequency, and the query length determine the absolute amount of work to be done in the scoring phase. The shape of the query-term frequency distribution may affect the efficiency of the algorithm. The database size and the frequency of non-stop-words determines the amount of storage required to store the inverted indexes and, again, the shape of the distribution affects the efficiency of the file layout. The database size, the average document length, and number of documents to be ranked affect the effort required to rank the documents once scored. The shape of the document-length distribution is probably not very important to these algorithms.

A database is synthesized by specifying the number of documents and size of the full text in Megabytes. If there are N_{doc} documents and b megabytes of data in the database, then (on the average) $\frac{bc_1}{i}$ occurrences of term i are generated and randomly assigned document identifiers between 0 and $N_{doc} - 1$. Queries are generated by a similar mechanism.

2.3 Data Parallel Computing

This paper assumes the data parallel programming model as described by Hillis and Steele (1986). The Connection Machine System, as described by Hillis (1985) and realized in the Connection Machine Model CM-2 by Thinking Machines Corporation (1987), will be used as the hardware model. The model includes a serial host computer and a large number of processing elements (PE's or processors). Data structures are distributed uniformly across the processing elements, and may be thought of as vectors. There are three aspects of computation: serial computation, local parallel computation, and non-local parallel computation. Serial computation consists of arbitrary operations on scalars. Scalars may be freely promoted to vectors by broadcasting their value to the processing elements. Local parallel computation consists of arbitrary element-wise computations applied to the contents of the PE's memories. In this mode, the processing elements may be thought of as a set of independent machines operating on scalar quantities stored in their local memories. Processing elements may temporarily deactivate themselves. Non-local parallel computation consists of operations that move data either from one processor to another or from the processing elements to the host. Non-local computations may be as simple as permuting the data or as complex as sorting it. Systems having between 4096 and 65,536 processing elements will be considered in this paper.

Each processor has its own local memory. Connection Machines may be configured with up to 128 Kilobytes per processor; this translates into 1/2 Gigabyte for a 4K processor machine or 8 Gigabytes for a 64K machine. Processors have indirect addressing hardware which allows them to access their local memory via an index register.² In addition, the indirect addressing hardware permits a group of 32 processors (called a *node*) to share each other's memory. For analytic purposes, one may sometimes treat a node as a single processor, so that a 64K processor machine might be thought of as a machine with 2048 *nodes*, each of which is itself a 32-processor shared-memory parallel SIMD machine. This point-of-view will frequently be taken in the discussion below.

I/O is provided by a high-throughput disk system. The Connection Machine uses a parallel disk array called the Data VaultTM mass storage system. Each such unit provides up to 40 Gigabytes of storage with a transfer rate of 25 Megabytes per second. Data Vault files are vector-structured: each location in the file stores one byte/word from each processing element in the machine. The datavault may be used in two modes: *striped mode* in which the datavault simulates a single disk with a very high transfer rate by striping data across all drives; and *independent access mode* in which all 32 drives may be accessed independently to simulate a disk-farm with a slightly lower transfer rate but a greater number of transfers-per-second. Up to 8 Data Vaults may simultaneously be active, providing transfer rates of up to 200 Megabytes per second.

3. Adapting Inverted Files to Parallel Computing

This section will present a simple implementation of inverted files on serial machines, then explain its adaptation to parallel computing.

3.1 Serial Inverted Indexes

Perparatory to considering parallel inverted indexes, it is worth considering the implementation of inverted indexes on serial machines, and the performance of such an implementation. A reasonably

2. This capability was missing from some early SIMD parallel machines, such as the Connection Machine model CM-1, Active Memory Technology's DAP (q.v. Flanders, 1977) and the Goodyear MPP (q.v. Batcher, 1980). In these machines, all processors were forced to access the same memory address on a given instruction cycle.

simple implementation, similar to that described by Doszkocs (1982), will be presented and its performance analyzed.

This paper presumes that the database has been indexed, and each document reduced to a set of structures of the form $\langle \text{document-id}, \text{term-id}, \text{weight} \rangle$ where *document-id* is an integer uniquely identifying a document; *term-id* is an integer uniquely identifying a term; and *weight* is a number representing the importance of the term. This structure is referred to as a *posting*.

Consider, for example, the set of documents shown in Figure 1. It consists of four documents, each containing between three and five words. The first step in indexing it would be to assign a term identifier to each of the eleven different words in the document set,³ as is done in Figure 2. Similarly, each document may be assigned an identifier, starting at 0. Weights are assigned by some suitable indexing procedure. In this case, each term will receive a weight of 1 which, when the document-vectors are normalized, will become $\frac{1}{\|D\|}$. The result the set of *raw postings* shown in Figure 3.

This is the first document	This be document two	I am doc- ument three	I am fourth
----------------------------------	----------------------------	-----------------------------	----------------

Figure 1: Four sample documents used in examples throughout this paper.

0	<i>am</i>	6	<i>is</i>
1	<i>be</i>	7	<i>the</i>
2	<i>document</i>	8	<i>this</i>
3	<i>first</i>	9	<i>three</i>
4	<i>fourth</i>	10	<i>two</i>
5	<i>I</i>		

Figure 2: Assignment of terms to term identifiers

Position	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Doc ID	0	0	0	0	0	1	1	1	1	2	2	2	2	3	3	3
Term ID	8	6	7	3	2	8	1	2	10	5	0	2	9	5	0	4
Weight	.20	.20	.20	.20	.20	.25	.25	.25	.25	.25	.35	.35	.35	.33	.33	.33

Figure 3: Raw Posting File for the documents in Figure 1

The first step in producing an inverted file is sorting these raw postings by term identifier (Figure 4). The positions of the first and last occurrences of each term are next recorded in a table called the *data map*, which will also include the mapping from text strings to term identifiers (Figure 5). The term identifiers in the postings are now redundant and are dropped. The result is a *inverted file*, which is stored on disk (Figure 6).

- Many of these words, such as *am* and *I* are stop words and would normally be dropped. They are retained here for the sake of the example.

Position	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Doc ID	2	3	1	0	1	2	0	3	2	3	0	0	0	1	2	1
Term ID	0	0	1	2	2	2	3	4	5	5	6	7	8	8	9	10
Weight	.25	.33	.25	.20	.25	.25	.20	.33	.25	.33	.20	.20	.20	.25	.25	.25

Figure 4: Sorted Postings

ID	Word	Start	End	ID	Word	Start	End
0	<i>am</i>	0	1	6	<i>is</i>	10	10
1	<i>be</i>	2	2	7	<i>the</i>	11	11
2	<i>document</i>	3	5	8	<i>this</i>	12	13
3	<i>first</i>	6	6	9	<i>three</i>	14	14
4	<i>fourth</i>	7	7	10	<i>two</i>	15	15
5	<i>I</i>	8	9				

Figure 5: Data Map

Position	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Doc ID	2	3	1	0	1	2	0	3	2	3	0	0	0	1	2	1
Weight	.25	.33	.25	.20	.25	.25	.20	.33	.25	.33	.20	.20	.20	.25	.25	.25

Figure 6: Inverted File

A query consists of a set of terms and weights, for example $3 * \text{document} + 2 * \text{this}$. Evaluating a query takes place in three stages: initialization, scoring, and ranking. Initialization consists of allocating one score register (called a *mailbox*) for each document and zeroing it (Figure 7). Next, the data-map entry for each term is accessed, so that the positions of the first and last postings for each query-term are known. In the query shown above, the term *document* occupies positions 3 through 5, and the term *this* occupies positions 12–13. The appropriate portions of the posting file are then brought into primary memory (Figure 8). The final step is to iterate through the query-terms, and through the postings for each query-term. The query-term weights are multiplied by the document-term weights (found in the postings), and used to increment the score in the mailbox indexed by the posting's document identifier. The results are shown in Figure 9.

0	0	0	0
---	---	---	---

Figure 7: Mailboxes initialized to 0

Term	Q-Weight	Postings
<i>Document</i>	3	<0, .20> <1, .25> <2, .25>
<i>This</i>	2	<0, .20> <1, .25>

Figure 8: Postings for each query term are loaded into memory

1.0	1.25	.75	0
-----	------	-----	---

Figure 9: Results of Query on Serial File

Finally, the mailboxes are scanned to locate the N_{ret} highest-ranking documents. This paper assumes $N_{ret} = 20$. A good serial algorithm may be described as follows:

- 1) The scores are divided into n groups of $\frac{N_{docs}}{n}$ documents..
- 2) The largest element of each group is located, placed in an array, and the original is zero'ed out.
- 3) The array is converted to a heap.
- 4) The first element of the heap is extracted. It is the highest-scoring document in the database.
- 5) The group from which it came is determined. That group is re-scanned, and a new group maximum is found.
- 6) The maximal element is put in the heap, and the original is zeroed.
- 7) Steps 4-6 are repeated until N_{ret} documents have been found.

This algorithm requires the construction of an n -element heap (step 3) and N_{ret} insert/delete operations (step 6). The time for the heap operations is thus $(n + N_{ret}) \log n$, and is clearly minimized by making n as small as possible. The algorithm also requires $n + N_{ret}$ scans of groups of $\frac{N_{docs}}{n}$ mailboxes (steps 2 and 5). The total time for the scanning is thus $(n + N_{ret}) \frac{N_{docs}}{n} = N_{docs} + \frac{N_{ret} N_{docs}}{n}$, and is clearly minimized by making n as large as possible. An optimal value for n may be found either analytically or empirically.

3.2 Performance of the Serial Algorithm

It is useful to measure the performance of the serial algorithm as a baseline in judging the performance of the parallel algorithms. The scoring algorithm was benchmarked by creating an array of 100,000 postings and an array of 100,000 mailboxes; the scoring algorithm was then applied to all 100,000 postings. The time measured on a fast serial processor (a Sun 4/330) was 0.78 seconds, giving a processing rate of .13 million postings per second.⁴ The ranking algorithm was also measured on a Sun 4/330. With $n = 1000$ (which was empirically determined to be optimal in this case), finding the highest-ranking 20 documents from a collection of 100,000 mailboxes required 0.11 seconds, giving a ranking rate of .91 million mailboxes per second. The CPU performance may then be extrapolated to various database sizes by computing, for each database size the total number of postings and total number of mailboxes to be scanned for each query. Table 2 presents performance estimates for databases between 1 and 128 Gigabytes, for queries of 10 and 30 terms. It must be noted that the figures below assume a main-memory database; no allowance is being made for the I/O time required to load postings into memory. Thus, actual application-level times are likely to be somewhat greater than predicted by the model below. These figures do, however, provide a meaningful lower bound on the time to score and rank documents on at least one high-performance serial machine. Other serial machines might, of course, be either faster or slower, and in particular it is quite likely that a more advanced microprocessor or a mainframe might achieve significantly higher performance.

4. Each posting was packed into a 32-bit quantity (8 bits weight, 24 bits document identifier); unpacking this quantity added to the run time. Higher performance might be obtained by using larger, unpacked postings, but this is generally a poor tradeoff, since memory capacity (for main-memory databases) and I/O bandwidth (for disk-resident databases) are generally more constraining than pure CPU performance in a well-balanced system.

File size, Gigabytes	1	2	4	8	16
Postings/Term (1000's)	3	6	12	24	48
Score 1 Term (msec)	23	46	92	184	368
Score 10 Terms (msec)	230	460	920	1840	3680
Score 30 Terms (msec)	690	1380	2760	5520	***
Documents (millions)	.2	.4	.8	1.6	3.2
Rank (msec)	219	438	876	1752	3504
Total, 10 terms (msec)	449	898	1796	3952	7184
Total, 30 terms (msec)	909	1818	3636	7272	***

Table 2: Performance of the serial algorithm on a Sun 4/330,

Some improvements in the performance of the ranking-portion of the serial algorithm may be obtained by pruning strategies. Such strategies are based on the fact that documents which match only high-frequency (hence low-selectivity) terms in a query are unlikely to be relevant to the user's request. According to Harmon (1990), pruning may reduce the number of documents to be ranked by 60% without significantly affecting the results of the search. More elaborate pruning schemes have been proposed by Buckley and Lewit (1985). The parallel algorithms described below might benefit from such pruning, but such modifications will not be considered in this paper.

3.3 Parallel Inverted Indexes

Stanfill, Waltz, and Thau (1989) describe a parallel algorithm that takes the posting file, in the form described above, and places it in a Connection Machine so that adjacent entries in the posting file are in adjacent processors. For example mapping the posting file in Figure 6 to a 4-processor machine result in 4 postings being placed in each processor (Figure 10). Mailboxes are assigned so that the mailboxes for consecutive documents map to consecutive processors (Figure 11).

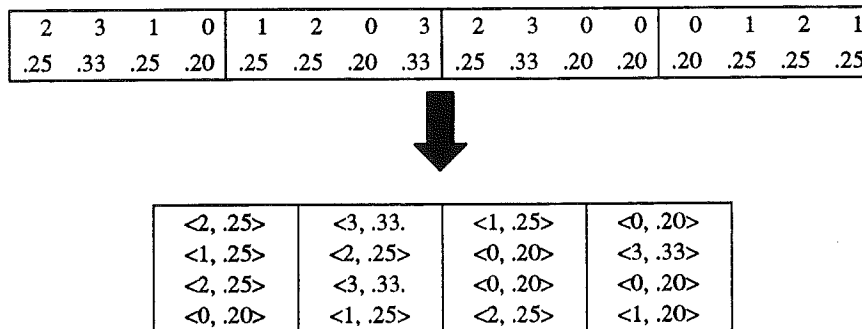


Figure 10: Assignment of Postings to Processors

0	0	0	0
---	---	---	---

Figure 11: Assignment of Mailboxes to Processors

To start processing the query, the location of the postings for each query term must be determined. The data map (Figure 5) is accessed as before to determine the file-offset for the first and last postings

of each term. On a machine with p processors, location n of the serial posting file maps to memory location $\left\lfloor \frac{n}{p} \right\rfloor$ in processor $n \bmod p$. In the above example, the postings for *document* occupy positions 3 through 5 of the serial posting file. These locations map to row 0 processor 3 through row 1 processor 2 of the parallel file. These location-ranges are then broken into groups that do not span row boundaries (Figure 12).

Word	Weight	Row	Start	End
<i>Document</i>	3	0	3	3
	3	1	0	1
<i>This</i>	2	3	0	1

Figure 12: Mapping query terms to non-spanning groups of postings

The I/O system is then called on to move these postings into memory. As noted above in the section on parallel computing, the I/O system operates by writing vectors (rows of data) to disk, and reading vectors into memory. The algorithm then iterates through the rows of this table. For each <weight, row, start, end> quadruple, those processors having processor ID's between *start* and *end* will access the posting at memory location *row*. The query-weight will be multiplied by the posting-weight. The result will be used to increment the contents of the appropriate mailbox. This last step involves sending data between processors.

In this example, the postings for *document* occupy processor 3 of row 0, and processors 0 and 1 of row 1. Execution starts by considering row 0 (Figure 13). First, the algorithm notes that only processor 3 contains row-0 postings for *document*; all other processors are deactivated. Second, this processor accesses the DOC ID's and document weights of row 0. Third, the document term-weight (.20) is multiplied by the query term-weight (3), yielding the posting's contribution to its document's score (.60). Finally, the processor sends a message to processor 0, telling it to add its contribution (.6) to the mailbox for document 0.

Execution now considers row 1 (Figure 14). First, the algorithm notes that only processors 0 and 1 contain row-1 postings for *document*; once again all other processors are deactivated. Second, DOC ID's and document weights for row 1 are accessed. Third, the document weight is multiplied by the query weight. Fourth, each processor determines the location of its document's mailbox. Finally, each processor sends a message to increment the proper mailbox.

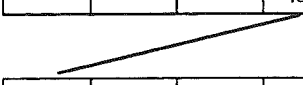
Processor ID	0	1	2	3
Doc ID's, Row 0	***	***	***	0
Weights, Row 0	***	***	***	.20
Query Weight	***	***	***	3
Posting weight * Query weight	***	***	***	.60
Send with Add				
Mailboxes Before	0	0	0	0
Mailboxes After	.6	0	0	0

Figure 13: Handling the postings in row 0

Processor ID	0	1	2	3
Doc ID's, Row 1	1	2	***	***
Query Weight	3	3	***	***
Weights, Row 1	.25	.25	***	***
Posting weight * Query weight	.75	.75	***	***
Send with Add				
Mailboxes Before	.60	0	0	0
Mailboxes After	.60	.75	.75	0

Figure 14: Handling the postings in row 1

On the Connection Machine, incrementing the mailboxes is accomplished via an instruction called *send-with-add*. This operation is the dominant cost of this algorithm, each such step taking 3 milliseconds.⁵ The algorithm just described yields good performance, but experience has shown that such inner-loop communication operations should, where feasible, be eliminated in order to get the best possible performance out of the system. A new file structure, *partitioned posting files*, will now be introduced to accomplish this purpose.

4. Partitioned Posting Files

The inner-loop *send-with-add* instruction is fundamentally unnecessary because document scoring is completely decomposable: if one has N documents, the scoring operation may be accomplished by N processors in unit time, with no communication. The original signature-based algorithm was implemented in essentially that manner, with each virtual processor being assigned a single document. The goal, then, is to find a representation which has the desirable properties of both the inverted file algorithm (suitability for use with external storage, support for document-term weights) and the signature-algorithm (locality). This section will arrive at such a data structure by starting with a posting file (essential for support of document-term weights), arranging the data so as to eliminate query-time communication, and finally partitioning the data so as to arrive at a structure suitable for use with external storage.

The *send-with-add* step may be eliminated if, rather than storing the postings in sequential order and *send'ing* them to the correct processor at query-time, they are stored in the same node⁶ as their mailbox. This file structure starts with the same set of tokens used to construct serial inverted indexes. Each document is mapped to a node. For example, given a machine with two processors, documents 0 and 2 might be assigned to node 0, and documents 1 and 3 might be assigned to node 1. The tokens are then moved to an arbitrary processor within that node, and sorted by ascending term ID within the node's shared memory (Figure 15).

5. Stanfill, Waltz, and Thau (1989) estimated the time to perform this operation at 1 millisecond. Subsequent benchmarking revealed this estimate to have been low.
6. As defined above, a *node* is a group of processors sharing common memory. In the case of the Connection Machine, a node is a group of 32 processors. In the case of machines with no shared memory, a node is a single processor. In the case of a machine with only shared memory, the entire machine may be considered a single node, although if there is both fast access to local memory and slower access to shared memory it may be best to ignore the shared memory and consider each processor as one node.

0	0	0	0	0	1	1	1	1	2	2	2	2	3	3	3
8	6	7	3	2	8	1	2	10	5	0	2	9	5	0	4
.20	.20	.20	.20	.20	.25	.25	.25	.25	.25	.25	.25	.25	.33	.33	.33



Row	Node 0			Node 1		
	Term ID	Doc ID	Weight	Term ID	Doc ID	Weight
0	0	2	.25	0	3	.33
1	2	0	.20	1	1	.25
2	2	2	.25	2	1	.25
3	3	0	.20	4	3	.33
4	5	2	.25	5	3	.33
5	6	0	.20	8	1	.25
6	7	0	.20	10	1	.25
7	8	0	.20			
8	9	2	.25			

Figure 15: Distributing postings to their home node

A problem with the representation at this point is that the postings for a given term are not guaranteed to be in the same row of data. This is called *skewing*. For example, the postings for term 8 are located in row 7 of node 0 and row 5 of node 1. This causes great difficulty when databases resident on secondary storage are constructed, since all rows of postings from the first occurrence of a term through last must be transferred to memory. Thus, in the example above rows 5 through 7 would need to be read. Unless countermeasures are taken, this skewing grows without bound as the database size increases.

Skewing cannot be eliminated, but it can be kept within bounds by *partitioning* the posting file. Each partition has a lower bound and an upper bound. Each partition contains only postings with word identifiers between those two bounds, inclusively. Furthermore, the upper bound of each partition is no greater than the lower bound of the next. For example, the database above might be divided into five partitions with bounds (0 ... 2) (2 ... 4) (5 ... 6) (7 ... 8) and (9 ... 10). Partition boundaries are stored on the host computer as part of the data map. Partitioning is, in essence a mechanism for forcing the periodic introduction of empty space in the posting file so as to retain a degree of alignment.

To create a partitioned posting file, it is necessary to select a blocking factor l , which will be the number of postings per partition, per node. Let l be this blocking factor (in the present example $l = 2$). An arbitrary processor in each node accesses the node's l 'th posting; this is called a *candidate boundary value*. The host then uses a *global minimum* operation to determine the smallest candidate boundary value; this becomes the *upper bound* (U) for a new partition. All postings in rows 0 through $l-1$ with term ID $\leq U$ are moved into the new partition. Remembering that the postings are sorted in ascending order, and that the upper bound is the smallest element in row l , it can be seen that the residue file contains all postings with ID $> U$, and the new partition contains all postings with ID $< U$. New partitions are created until no data remains.

For the example in Figure 15, row 2 of the posting file contains the term ID's 2 and 2. These are the candidate boundary values. The boundary value is then also 2. All postings in rows 0 and 1 of the

unpartitioned file which are no greater than 2 are transferred to the partitioned file, and the bounds are noted. This yields the new partition shown in Figure 16, and the residue of unconsumed postings shown in Figure 17. This process is repeated until all postings have been transferred. The result is the partitioned posting file shown in Figure 18. It is immediately obvious that the data structure in Figure 18 does not fully utilize the available space: the file contains sufficient space for 18 postings, of which 16 are actually used, for a utilization of 89%.

Row	Bounds		Node 0			Node 1		
	Low	High	Term ID	Doc ID	Weight	Term ID	Doc ID	Weight
0	0	2	0	2	.25	0	3	.33
1			2	0	.20	1	1	.25

Figure 16: The first partition, with associated low and high bounds.

Row	Node 0			Node 1		
	Term ID	Doc ID	Weight	Term ID	Doc ID	Weight
2	2	2	.25	2	1	.25
3	3	0	.20	4	3	.33
4	5	2	.25	5	3	.33
5	6	0	.20	8	1	.25
6	7	0	.20	10	1	.25
7	8	0	.20			
8	9	2	.25			

Figure 17: Unconsumed postings after the first partition is created.

Row	Bounds		Node 0			Node 1		
	Low	High	Term ID	Doc ID	Weight	Term ID	Doc ID	Weight
0	0	2	0	2	.25	0	3	.33
1			2	0	.20	1	1	.25
2	2	4	2	2	.25	2	1	.25
3			3	0	.20	4	3	.33
4	5	6	5	2	.25	5	3	.33
5			6	0	.20			
6	7	8	7	0	.20	8	1	.25
7			8	0	.20			
8	9	10	9	2	.25	10	1	.25

Figure 18: Partitioned posting file

In the serial and parallel implementations of inverted files, a data map precisely defines which postings are associated with which terms. The term identifiers are thus unneeded and are dropped. With partitioned posting files, the upper and lower bounds of the partition do not uniquely determine the term associated with each posting. Thus the term identifiers cannot simply be dropped. Naively, one might simply retain the term identifier in the posting file. Note, however, that each partition contains a limited number of different terms, which allows the representation of term identifiers with short integers. If, for example, no partition contains more than 32 distinct terms, then 5 bits suffice to uniquely distinguish the terms within a partition. Such a compressed term identifier will be referred to as a *term tag*. The document identifier can also be compressed. If there are D documents and N

nodes, then there are $\frac{D}{N}$ documents per node, and a document may be uniquely represented by

$\left\lceil \log_2 \frac{D}{N} \right\rceil$ bits. This compressed document identifier will be called a *document tag*. Finally, one

may (generously) allocate 16 bits to the weight contained in the posting. The result is as shown in Figure 19.

Row	Bounds		Node 0			Node 1		
	Low	High	Term Tag	Doc Tag	Weight	Term Tag	Doc Tag	Weight
0	0	2	0	1	.25	0	1	.33
1			2	0	.20	1	0	.25
2	2	4	0	1	.25	0	0	.25
3			1	0	.20	2	1	.33
4	5	6	0	1	.25	0	1	.33
5			1	0	.20			
6	7	8	0	0	.20	1	0	.25
7			1	0	.20			
8	9	10	0	1	.25	1	0	.25

Figure 19: Compressed Partitioned Posting File

The scalar portion of this data structure resides in the host's primary memory; the vector data will generally be stored on parallel secondary storage, and moved into memory one partition at a time. A data map (e.g. a hash table) mapping terms to term ID's and the range of partitions containing postings for that term completes the data structure.

4.1 Storage Requirements

Three factors determine storage requirements: the fraction of empty space in the partitioned posting file, the number bits in the term-tag, and the number of bits in the document-tag. The number of bits in the document tag was derived above. The number of bits in the term tag and the fraction of empty space in the posting file may be estimated by simulation. The simulation procedure is as follows:

- 1) A random non-stop-word term T_i is selected.
- 2) Its frequency is determined by equation 2.
- 3) Its frequency is multiplied by the size of that database to yield the number of occurrences in the database.
- 4) These occurrences are distributed at random among 256 nodes (corresponding to an 8K-processor machine).
- 5) Steps 1-4 are repeated until sufficient data has been generated to yield statistically significant results.
- 6) The resulting set of postings are partitioned. The partition boundaries are recorded, as are the number of empty positions. A block-size of 128 postings per partition/node is employed.
- 7) The utilization (% of available posting slots actually occupied) is computed.
- 8) The number of bits in the document- and term- tags are computed. The size of the posting is computed assuming 16 bits for weights.

- 9) The total number of postings in the database is determined by dividing by the frequency of non-stop-words (1 per 17.3 bytes in this example).
- 10) The total space is obtained by multiplying the number of postings by the size of the postings, divided by the utilization.
- 11) The overhead is obtained by dividing the posting-file size by the size of the database.
- 12) For machines larger than 256 nodes, it is assumed that doubling the quantity of data and the number of processors keeps the occupancy and posting size constant, leaving storage overhead unchanged.

Table 3 shows the results of this simulation for databases between 1 and 1024 Gigabytes, assuming a 256 node (8K processor) machine. Equivalent database sizes for machines having 512 (16KP), 1024 (32KP) and 2048 (64KP) are also shown. Storage overhead is remarkably constant over a huge range of database size, with improved occupancy being exactly balanced by growth in the posting size.

File size in Gigabytes	1	2	4	8	16	32	64	128	256	512	1024
Equivalent on 512 nodes	2	4	8	16	32	64	128	256	512	1024	2048
Equivalent on 1024 nodes	4	8	16	32	64	128	256	512	1024	2048	4096
Equivalent on 2048 nodes	8	16	32	64	128	256	512	1024	2048	4096	8192
Occupancy	81	82	84	85	87	89	91	93	94	95	97
Max Words/Partition	180	99	56	36	19	14	8	5	3	3	2
Term Tag Bits	8	7	6	6	5	4	4	3	2	2	2
Document Tag Bits	11	12	13	14	15	16	17	18	19	20	21
Posting Size in Bits	35	35	35	36	36	36	37	37	37	38	39
Total Postings (billion)	.057	.12	.23	.46	.93	1.8	3.7	7.4	15.8	39.6	59.2
Posting File Size (GB)	.31	.62	1.2	2.4	4.8	9.4	18.8	36.8	72.8	148.0	297.5
Overhead	31%	31%	30%	30%	30%	29%	29%	29%	28%	29%	29%

Table 3: Storage utilization

As noted above, a 256 node (8KP) system has 1 Gigabyte of primary memory. Under that assumption, a database 3 Gigabytes would fit in primary memory and still leave 100 Megabytes of scratch space. In addition, it has been noted that a Datavault can hold up to 40 Gigabytes of data, so that a single datavault could hold the postings for a database of 120 Gigabytes. A 1024 Gigabyte database would require eight Datavaults.

5. The Scoring Algorithm

Given this data structure, queries can be executed without moving data between nodes. First, each processor allocates and zeros-out one mailbox for each document it has been assigned⁷(Figure 20). Second, the data map is consulted to determine which partitions need to be loaded into primary memory. The tags corresponding to each term are then determined (Figure 21). The first partition required for the query (partition 0) is then loaded into primary memory (Figure 22). Each processor then loops through its postings, looking for postings with the correct term tag (2 at this point). When

7. On the CM each processor is assigned 1/32 of the documents for its node.

one is found, the document tag is used to determine which of the mailboxes in the shared memory of the node is to be accessed. In the present example, a processor in node 0 finds a matching term tag in row 2. This posting has document tag 0, corresponding to mailbox 0. At this point, the query weight (3) is multiplied by the document weight (.20), and the product used to increment the appropriate mailbox (Figure 23). Note that, because duplicate occurrences of a term within a document are forbidden, and only one term is processed at a time, there is no danger of two processors attempting to access the same mailbox.

Mailboxes	0	0
	0	0

Figure 20: Two mailboxes per node, initialized to 0

Term	Weight	partition	Term ID	Low Bound	Tag
<i>Document</i>	3	0	2	0	2
		1	2	2	0
<i>This</i>	2	4	8	8	0

Figure 21: A list of all partitions needed for a query, plus data-map information

Node 0			Node 1		
Term Tag	Doc Tag	Weights	Term Tag	Doc Tag	Weights
0	1	.25	0	1	.33
2	0	.20	1	0	.25

Figure 22: Postings from partition 0

Mailboxes	.60	0
	0	0

Figure 23: Mailboxes after first partition has been scored

The process is repeated as, in turn, each partition is loaded, searched for postings with the appropriate term tag, and the mailbox addressed by the document tag incremented. Continuing example, partition 1 would be loaded and searched for occurrences of term-tag 0 (Figure 24). Two such occurrences are found, both on row 0 of the partition. As a result, the scores for mailbox 1 in node 0, and mailbox 0 in node 1 are incremented (Figure 25). The process is repeated until all partitions for all query-terms have been processed.

Node 0			Node 1		
Term Tag	Doc Tag	Weights	Term Tag	Doc Tag	Weights
0	1	.25	0	0	.25
1	0	.20	2	1	.33

Figure 24: Partition 1 is the second partition for *document*

Mailboxes	.60	.75
	.75	0

6. The Ranking Algorithm

The output of the above algorithm is a set of mailboxes containing document scores. It is now necessary to identify and rank the documents with the highest scores. The following algorithm, due to Jim Hutchinson, is an efficient method for accomplishing this task. This algorithm uses the processor-wise model rather than the node-wise one.

The mailboxes may be viewed as a 2-dimensional array, with rows corresponding to locations within processor memory and columns corresponding to processors. To illustrate this aspect of the system, a new example will be used, in which there are 4 processors and 16 documents, yielding 4 mailboxes per processor (Figure 26).

3	8	16	7
99	4	34	87
18	97	2	1
0	45	18	6

Figure 26: A new sample problem, with 16 mailboxes in 4 processors

The first step is to determine the document identifier corresponding to each mailbox. As noted above, the least-significant bits of a document ID select a node, and the most significant bits become the document tag, which is used to select a mailbox within a processor's memory. This implies the correspondence between mailboxes and document ID's shown in Figure 27. The document identifier is then appended to the score, producing an array of surrogates (Figure 28). Next, the largest element in each row is determined using a *global maximum* operation, the results of which are stored on the serial host. As these maxima are computed, they are stored in an array on the scalar front-end and the original data is zero'ed out (Figure 29). The maxima are then converted to a heap. The first element of the heap is then extracted. In this case, the maximum element is 99.04, which indicates that document 4 had a score of 99. The maximum for its row is then re-computed (Figure 30).

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Figure 27: Document Identifiers

3.00	8.01	16.02	7.03
99.04	4.05	34.06	87.07
18.08	97.09	2.10	1.11
0.12	45.13	18.14	6.15

Figure 28: Surrogates (score.docid)

max	Surrogates			
16.02	3.00	8.01	0	7.03
99.04	0	4.05	34.06	87.07
97.09	18.08	0	2.10	1.11
45.13	0.12	0	18.14	6.15

Figure 29: Maximal surrogates

max	Surrogates			
16.02	3.00	8.01	0	7.03
87.07	0	4.05	34.06	0
97.09	18.08	0	2.10	1.11
45.13	0.12	0	18.14	6.15

Figure 30: Surrogates after first document is extracted

This process is repeated until sufficient documents have been found. The cost is one global maximum and one heap-insert operation per row of mailboxes, plus one global maximum, one heap insert, and one heap-delete operation per document to be ranked. The time required for these operations is dominated by the time to compute the global-maxima.

6.1 Performance

The performance of the ranking algorithm was determined by the following procedure:

- 1) The number of documents is determined by dividing the size of the database by the average document size (5000 bytes).
- 2) A sufficient number of mailboxes are allocated.
- 3) The mailboxes are filled with random numbers.
- 4) The 20 highest-ranking scores are extracted from the machine.

For databases between 1 and 1024 Gigabytes and a 256 node machine, the times are as reported in Table 5. Again, database sizes corresponding to larger machines are shown.

File Size, Gigabytes	1	2	4	8	16	32	64	128	256	512	1024
Equivalent on 512 nodes	2	4	8	16	32	64	128	256	512	1024	2048
Equivalent on 1024 nodes	4	8	16	32	64	128	256	512	1024	2048	4096
Equivalent on 2048 nodes	8	16	32	64	128	256	512	1024	2048	4096	8192
Ranking Time (msec)	27	37	53	80	132	278	410	781	1504	3000	6000

Table 5: Estimated times for ranking documents after scoring

The largest database involves a 8192 Gigabytes of data, which would contain 1.6 billion documents. The ranking time of 6 seconds on a 2048 node machine thus corresponds to 267 million mailboxes per second. The Sun 4/330 delivered a performance of .91 million mailboxes per second, so the parallel algorithm seems to deliver 293 times higher performance. These measurements were performed on a machine having 256K bits per processor, rather than the maximal configuration of 1M bits per processor. This provided insufficient mailbox space for the 512- and 1024- Gigabyte databases. Times for those databases were, therefore, extrapolated.

7. Secondary Storage

Before discussing the adaptation of partitioned posting files to secondary storage, it is useful to understand the architecture of disk systems for parallel computers. Paterson (1988) has suggested

that, in order to support the high I/O rates required by parallel computers while maintaining a high degree of reliability, it is sufficient to construct disk arrays from large numbers of inexpensive disks, provided redundant information is stored to guard against media failures. Given n disks, each with a transfer rate of T , one may construct a disk array with a transfer rate of nT , which is used as if it were a single disk (a technique called *disk striping* by Salem (1986)). The Connection Machine has available a disk array called the *Datavault*, which has either 32 or 64 data drives, and supports a transfer rate of up to 25 megabytes per second. In order to guard against loss of data due to drive failure, disk arrays generally contain some extra disk units and keep a certain amount of redundant information, but this is transparent as far as the user is concerned. To provide still higher transfer rates, up to 8 Datavaults may be combined in parallel, providing transfer rates as high as 200 megabytes per second. The I/O system still appears to the user as a single disk unit with a very high transfer rate. The number of simultaneously active Datavaults will be referred to as the *striping factor*.

The read- and write- operations on a disk array will typically transfer information to/from all processors at once, moving data to/from the same location in each processor (Figure 31). This is well suited to the partitioned posting file representation described above: in a single I/O operation, it is possible to transfer a contiguous set of partitions into CM memory. Thus, the query processing strategy is to 1) determine which partitions are needed; 2) transfer only those partitions from secondary to primary storage; and 3) use the scoring and ranking algorithms as described above.

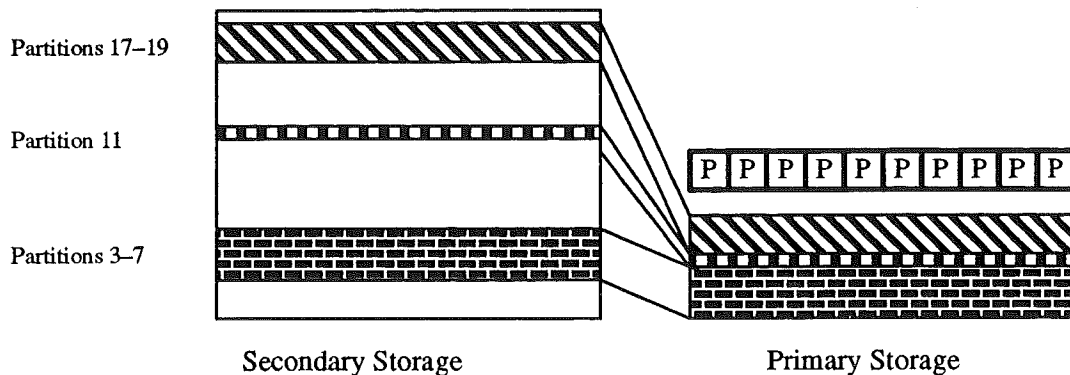


Figure 31: Striped disk access

7.1 Performance

The Datavault has a latency of 200 milliseconds and a transfer rate of 25 megabytes per second. It is possible to overlap computation with latency, but not with transfers. The performance of the system may be determined as follows:

- 1) Determine the average size of a partition in bytes by multiplying the posting-size in bits (q.v. Table 3) by the number of slots per partition (32,768).
- 2) Multiply this by the average number of partitions per query-term (q.v. Table 4).
- 3) Determine the transfer time by dividing this figure by the transfer rate of 25 MB/second.
- 4) The latency is always 200 milliseconds.

Latency and transfer times are estimated for databases between 1 and 1024 Gigabytes (Table 6), assuming a striping factor of 1. Corresponding times for striping factors of 2, 4, and 8 are also shown.

It is immediately apparent that, for databases of moderate size (1–128 Gigabytes), the latency dwarfs the transfer time. A method for improving on this situation is clearly desirable.

File Size, Gigabytes	1	2	4	8	16	32	64	128	256	512	1024
Equivalent with stripe = 2	2	4	8	16	32	64	128	256	512	1024	2048
Equivalent with stripe = 4	4	8	16	32	64	128	256	512	1024	2048	4096
Equivalent with stripe = 8	8	16	32	64	128	256	512	1024	2048	4096	8192
Posting Size in Bits	35	35	35	36	36	36	37	37	37	38	39
Partition Size, MB	.143	.143	.143	.147	.147	.147	.152	.152	.152	.156	.160
Partition/Term	1.1	1.2	1.4	1.9	2.7	4.3	7.5	13.6	26.0	50.5	100.0
Transfer/Term (MB)	.157	.172	.200	.280	.397	.632	1.14	2.07	3.95	7.89	16.0
Transfer Time/Term (msec)	6	7	8	11	16	25	46	83	158	316	640
Latency/Term (msec)	200	200	200	200	200	200	200	200	200	200	200

Table 6: I/O Behavior for striped datavaults

7.2 Independent Disk Access

The DataVault contains a large number of disks which, if harnessed, ought to provide a large number of I/O's per second, effectively reducing the latency. This is currently supported as an experimental feature of the Datavault. In this independent disk access mode, the Datavault must be thought of as a 2-dimensional array of disk blocks, where columns correspond to the individual disks and rows correspond to disk-addresses. In one primitive independent read operation, it is possible to read one block from each column (drive). The read operation will transfer information to all processors at once, spreading a disk-block evenly across all processors (Figure 32). Ultimately, some form of error correction/redundancy will be needed, such as one of Paterson's RAID schemes (1988).

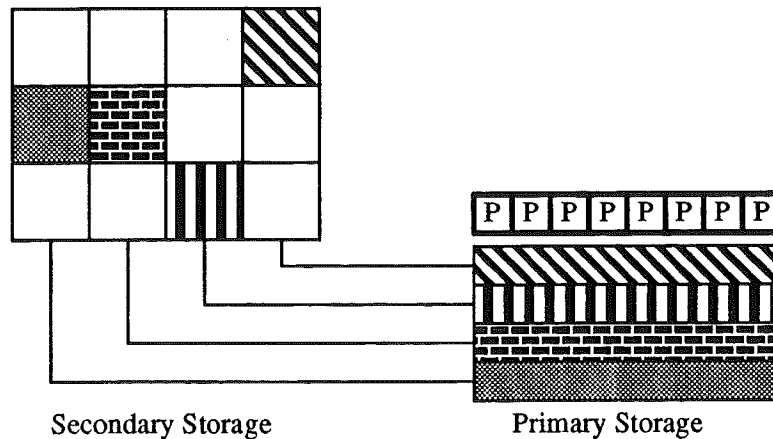


Figure 32: Independent Disk Mode

The partitioned posting file is mapped to this disk array in row-major order, so that row i , column j corresponds to partition $j + i * 32$ (assuming there are 32 disks).

The algorithm for accessing secondary storage is as follows:

- 1) When a query is received, the partitions needed to service the query are located.
- 2) These partitions are mapped to rows and columns in the disk array.
- 3) This set of row-column requests is given to the disk system, which parcels them out among the 32 disks, performing as many primitive independent-disk reads as are necessary to retrieve the needed information.
- 4) As each set of 32 partitions is read into memory, they are scored in accordance with the algorithm in Section 5.

The workloads assigned to the 32 disks will not be uniform. For example, if 32 partitions are to be retrieved from 32 disks, it is likely that several disks will have none of the needed partitions, while one disk might have 3 requests. The maximum number partitions mapping to a single disk will be called the *depth* of the query; it is the depth that governs the number of primitive independent disk reads, hence the time required to service the query.⁹

7.3 Performance with independent disk

It takes 100 milliseconds to initiate independent disk access mode. Once this has been done, each primitive independent disk read has an additional latency of 125 milliseconds, after which data will flow at 25 megabytes/second. After data has been transferred, the CM requires 18 milliseconds of compute time per megabyte of data (assuming a 256 node system) to rearrange the data. The latency period may be overlapped with the rearrangement period, and with arbitrary computations. The transfer period may not be overlapped with anything. The following computation establishes the basic performance characteristics of independent disk access for partitioned posting files:

a. Maximum Posting Size:	40 bits (5 bytes)
b. Partition capacity (256 node system):	32,768 postings
c. Partition Size ($a * b$)	163,840 bytes
d. Transfer Size ($32 * c$)	5.24 megabytes
e. Transfer Rate	25 megabytes/second
f. Transfer Time (d/e)	210 msec
g. Rearrangement Time Factor	18 msec / megabyte
h. Rearrangement Time ($g * d$)	94 msec
i. Time to score 1 partition	.695 msec
j. Time to score 32 partitions ($32 * i$)	22 msec
k. Rearrangement + scoring time ($h + j$)	116 msec
l. Latency	125 msec
m. Per-depth time ($f + \max(l, k)$)	335 msec

Accessing a batch of 32 partitions thus involves 210 msec of transfer time plus 125 msec of latency, which can be exactly overlapped with rearrangement and scoring, for a total per-phase time of 335 milliseconds. It remains only to determine the depth of the average query. This was done by the following procedure:

9. A consequence of mapping partitions to disks in round-robin order is that, in cases where a term involves multiple partitions, those partitions are guaranteed to map to different disks, a property which reduces load imbalance.

- 1) A query is created by selecting either 10 or 30 terms at random, according to the query-term distribution described above.
- 2) The required number of partitions required to service the request is determined by the procedure previously outlined.
- 3) The disk of the first partition for each query term is determined at random.
- 4) If a given query-term requires multiple partitions, those partitions are assigned to subsequent disks in round-robin order, as described above.
- 5) The maximum query-depth (greatest number of partitions mapping to a given disk) is determined.
- 6) Steps 1-4 are repeated until enough queries have been generated to yield statistically significant results.

The results are shown in Table 7. In this case, results do not necessarily scale uniformly with system size; it is not clear that an 8 GB database on a system with 8 Datavaults and 2048 nodes has the same performance characteristics as a 1 GB database with 1 Datavault and 256 nodes.

File Size, Gigabytes	1	2	4	8	16	32	64	128	256	512	1024
Query Depth, 10 terms	1.9	1.9	2.0	2.2	2.5	3.1	4.0	6.0	9.6	16.4	31.6
I/O + Score Time (sec)	.6	.6	.7	.7	.8	1.0	1.3	2.0	3.2	5.5	10.6
Query Depth, 30 terms	3.4	3.4	3.7	4.1	5.1	6.7	9.4	15.0	26.3	48.1	91.7
I/O + Score Time (sec)	1.1	1.1	1.2	1.4	1.7	2.2	3.1	5.0	8.8	16.1	30.7

Table 7: I/O plus score times for independent disk access system

8. Retrieval System Architectures

It is now time to combine the various results from other sections of this paper, and propose some architectures. Three basic architectures will be presented:

- 1) A main-memory architecture (no secondary storage access at run-time).
- 2) A disk-resident database using independent disk access.
- 3) A disk-resident database using striped access.

As will be seen, these systems are suitable for use on progressively larger databases.

8.1 The main-memory architecture

The main-memory database is limited by the capacity of primary memory. Depending on the size of machine employed, memory could be between .5 and 8 Gigabytes. As noted above, the partitioned posting file is 30% the size of the text file. Thus, on the largest available machine databases up to 24 Gigabytes may be stored, leaving 10% of memory free for scratch space and mailboxes. For this architecture, the number of processors, hence the available compute-power, is proportional to the size of the database, and response-times are constant. Because no I/O is needed, response is extremely fast. This architecture is attractive for a central server shared by a large number of subscribers, accessing databases of modest size but great value. As the price of semiconductor

memory continues to plummet, this architecture will become increasingly attractive. This architecture is summarized in Table 8.

File Size, Gigabytes	1.5	3	6	12	24
Posting File Size (GB)	.45	.9	1.8	3.6	7.2
Machine Size (nodes)	128	256	512	1024	2048
Machine Size (procs)	4K	8K	16K	32K	64K
Main Memory (GB)	.5	1	2	4	8
Scratch Space (GB)	.05	.1	.2	.4	.8
Documents in DB (million)	.3	.6	1.2	2.4	4.8
Memory for Mboxes (GB)	.001	.002	.004	.008	.016
Time/query-term (sec)	.001	.001	.001	.001	.001
Time/10 terms (sec)	.010	.010	.010	.010	.010
Time/30 terms (sec)	.030	.030	.030	.030	.030
Rank Time (sec)	.045	.045	.045	.045	.045
Total Time/10 terms (sec)	.055	.055	.055	.055	.055
Total Time/30 terms (sec)	.075	.075	.075	.075	.075

Table 8: Performance characteristics of main-memory architecture

Table 2 predicts a time of .449 seconds for a 10 term query on a 1 Gigabyte database, using a Sun 4/330, and a time of .909 seconds for a 30 term query on the same database. Extrapolating to a 24 Gigabyte database, one gets times of 10.8 seconds and 21.8 seconds. Dividing by the predicted performance of the main-memory architecture, one sees a performance gain of 195 in the first case and 316 in the second.

8.2 The Independent Disk Architecture

For databases which are of moderate size, and either too large to fit into the memory of any machine or not accessed sufficiently often to justify the expense of buying a large machine, a system using independent disk access is appropriate. Databases between 4 and 128 Gigabytes can be comfortably handled by a system with a single 40 Gigabyte datavault and a single 8K-processor (256 node) Connection Machine. The I/O and scoring times for this system have already been deduced; it remains only to add in the ranking times. The characteristics of this system are summarized in Table 9.

File Size, Gigabytes	4	8	16	32	64	128
Posting File Size, GB	1.2	2.4	4.8	9.6	19.2	38.4
Machine Size (nodes)	256	256	256	256	256	256
Machine Size, Procs	8K	8K	8K	8K	8K	8K
Datavaults	1	1	1	1	1	1
Query Depth, 10 terms	2.0	2.2	2.5	3.1	4.0	6.0
Scoring Time (sec)	.7	.7	.8	1.0	1.3	2.0
Query Depth, 30 terms	3.7	4.1	5.1	6.7	9.4	15.0
Scoring Time (sec)	1.2	1.4	1.7	2.2	3.1	5.0
Ranking Time (sec)	.053	.080	.132	.228	.410	.781
Total Time, 10 terms (sec)	.8	.8	.9	1.2	1.7	2.8
Total Time, 30 terms (sec)	1.3	1.5	1.8	2.4	3.5	5.8

Table 9: Performance characteristics of independent disk architecture

8.3 The Striped Disk Architecture

Once the size of the database exceeds 128 Gigabytes, transfers become large enough to justify the use of striped disk mode, which optimizes for transfer rate at the expense of latency. As the database grows, more Datavaults and/or more processors are added. It is assumed that the striping factor equals the number of 8KP CM segments in the system, so that a 64KP system would support a striping factor of 8. Additional datavaults are added as necessary to provide additional storage capacity. For a given size database, there is generally a tradeoff between the number of processors (hence the possible striping factor) and the system performance. Times for 10-term queries are shown in Table 10; times for 30-term queries will be correspondingly longer. In all cases, the time to score a query-term is much smaller than the latency, so the time to score a query-term is simply equal to the latency time plus the transfer time.

File Size, Gigabytes	128	256	512	1024	256	512	1024	2048
Posting File Size, GB	38	76	152	304	76	152	304	608
Machine Size (nodes)	256	512	1024	2048	256	512	1024	2048
Machine Size (procs)	8K	16K	32K	64K	8K	16K	32K	64K
Datavaults	1	2	4	8	2	4	8	16
Transfer Time (sec)	.083	.083	.083	.083	.158	.158	.158	.158
Latency (msec)	.200	.200	.200	.200	.200	.200	.200	.200
Scoring Time/Term (sec)	.009	.009	.009	.009	.018	.018	.018	.018
Query Time/Term (sec)	.283	.283	.283	.283	.358	.358	.358	.358
Query Time/10 Terms (sec)	2.8	2.8	2.8	2.8	3.6	3.6	3.6	3.6
Ranking Time (sec)	.8	.8	.8	.8	1.5	1.5	1.5	1.5
Total Time/10 Terms (sec)	3.6	3.6	3.6	3.6	5.1	5.1	5.1	5.1

File Size, Gigabytes	512	1024	2048	4096	1024	2048	4096	8192
Posting File Size, Gigabytes	152	304	608	1216	304	608	1216	2432
Machine size (nodes)	256	512	1024	2048	256	512	1024	2048
Machine size (procs)	8K	16K	32K	64K	8K	16K	32K	64K
Datavaults	4	8	16	32	8	16	32	64
Transfer Time (sec)	.316	.316	.316	.316	.640	.640	.640	.640
Latency (msec)	.200	.200	.200	.200	.200	.200	.200	.200
Scoring Time/Term (sec)	.035	.035	.035	.035	.068	.068	.068	.068
Query Time/Term (sec)	.516	.516	.516	.516	.840	.840	.840	.840
Query Time/10 Terms (sec)	5.2	5.2	5.2	5.2	8.4	8.4	8.4	8.4
Ranking Time (sec)	3.0	3.0	3.0	3.0	6.0	6.0	6.0	6.0
Total Time/10 Terms	8.2	8.2	8.2	8.2	12.4	12.4	12.4	12.4

Table 10: Performance characteristics of striped disk architecture

9. Discussion

Figure 33 summarizes the various architectural options, in terms of response time and database size. First, it should be noted that one of the three architectures is appropriate to almost any database that can be imagined, and that reasonable response times are possible for databases of essentially unlimited size. In an architectural sense, document ranking can thus be considered a solved problem. Doubtless there is room for improvement. Indeed, the above architectures go far beyond the actual demands of the day; text databases of more than a few hundred gigabytes are currently unknown.

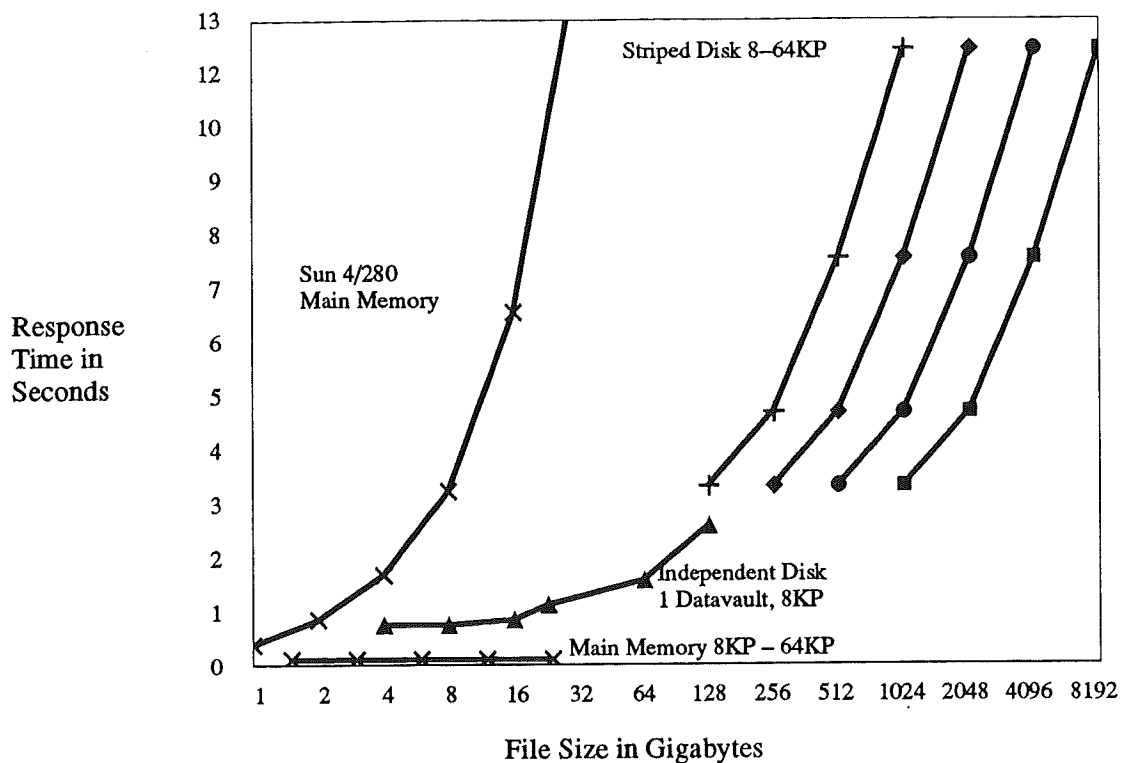


Figure 33: Summary of Architectures

The algorithms described above are modest in the demands they make on processor architecture. Very little communication is needed at run-time. A serial host must broadcast a small amount of information (term tags, query-term weights) to the processing elements every time a new partition is processed. The ranking algorithm primarily consists of global-maximum operations. The vast majority of interprocessor communication takes place when the partitioned posting file is constructed; further work is needed to determine how much communication is actually required. The processing elements themselves must support indirect addressing; this is not a problem with most contemporary parallel computers.

There is, however, considerable need for additional work. There remain some straight-forward algorithmic issues. For example, the problem of building and updating these databases has not yet been fully addressed; this is currently being studied. There also remain some substantial technological issues relating to storage technology. First, the main memory database architecture, with its near-instant response times, is highly desirable, but is restricted by current memory technology, which makes the assembly of a Connection Machine system having more than 8 Gigabytes of semiconductor memory impossible at this time. There is no doubt that, with the continuing rapid evolution of memory technology, this limitation will soon be overcome. Within this decade, databases of up to 128 Gigabytes should fit into primary memory. There is also need for more work on disk technology and disk array technology. At this writing, no Connection Machine has been configured with more than 3 Data Vaults. The largest configuration proposed above requires 64 Datavaults; the logistics of managing such a huge amount of storage remain unsolved. In addition, the cost of secondary storage remains, for the largest databases, a considerable problem. For an

organization that really needed access to a database of 8 Terabytes, the cost of a 64K-processor Connection Machine should not, in practice, be a major obstacle, but the cost of storage media would likely prove prohibitive. However, as optical and other low-cost, high-volume media mature, it is likely that this problem will be overcome. Finally, for databases of intermediate size, the latency and bandwidth characteristics of the DataVault, even with independent disk access, remain a limiting factor; improvements in these areas might prove quite helpful.

The above problems are either algorithmic or technological, and will certainly be solved. The primary problems at this point are those relating to the science of information retrieval, in particular automatic indexing. Parallel computing technology has evolved to the point that searching a 128 Gigabyte database does not present a significant technological problem. Predicting the quality of such a search, however, remains a considerable scientific challenge. The ability of serial machines to collect and search text has already strained the manual methods of library science. With parallel computing, the quantities of data that can be collected and searched are so staggering that there is no hope whatsoever of manually indexing it, leaving no alternative to full-text databases and automatic information retrieval. However, the majority of research in information retrieval has been conducted on tiny (< 100 Megabyte) databases of abstracts from scientific publications. The technology of text databases has thus outstripped the science of text databases by five orders of magnitude. This is a situation which, given suitable funding of basic research in IR, might be remedied, but in the absence of adequate support the application of the technology presented in this paper may be limited by a lack of knowledge of how to apply it.

REFERENCES

- Batcher, K. E. (1980). Design of a massively parallel processor. *IEEE Transactions on Computing*, C-29(9), 836-840.
- Buckley, C., and Lewit, A. (1985). Optimization of Inverted Vector Searches. Paper presented at the International Conference on Research and Development in Information Retrieval. Montreal, Canada.
- Croft, B. (1988). Implementing Ranking Strategies Using Text Signatures. *ACM Transactions on Office Information Systems*. 6(1), 42-62.
- Doszkocs, T. E. (1982) From Research to Application: The CITE Natural Language Information Retrieval System. In Salton G. and Schneider H.J. (Eds.), *Research and Development in Information Retrieval*, (pp. 251-262), Berlin: Springer-Verlag.
- Harman, D., (1990). Retrieving records from a Gigabyte of Text on a Minicomputer using Statistical Ranking. To appear, *Journal of the American Association for Information Science*.
- Flanders, P.M., et. al. (1977). Efficient high speed computing with the distributed array processor. In Kuch, Lawrie, and Sameh (Eds.), *High Speed Computing and Algorithm Organization* (pp. 113-127). New York: Academic Press.
- Hillis, D. (1985). *The Connection Machine*. Cambridge, MA: MIT Press.
- Paterson, D. A., Gibson, G., and Katz, R. H. (1988, June). A Case for Redundant Arrays of Inexpensive Disks (RAID). Paper presented at the ACM SIGMOD Conference on Management of Data. Chicago, Illinois.

- Pogue, C. and Willett, P. (1987). Use of Text Signatures for Document Retrieval in a Highly Parallel Environment. *Parallel Computing*, 4, 259-268.
- Rocchio, J. J. Jr. (1971). Relevance Feedback in Information Retrieval. In Salton, G. (Ed.), *The Smart System — Experiments in Automatic Document Processing*, 313-323. Englewood Cliffs, NJ: Prentice-Hall.
- Salem, K. and Garcia-Molina, H. (1986) Disk Striping. Paper presented at the IEEE International Conference on Data Engineering.
- Salton, G. (1970). Automatic Text Analysis. *Science*, 168, 335-343.
- Salton, G. (Ed.). (1971). *The Smart System — Experiments in Automatic Document Processing*. Englewood Cliffs, NJ: Prentice-Hall.
- Salton, G., Wong, A, and Yang, C.S. (1975). A Vector Space Model for Automatic Indexing. *Communications of the ACM*, 18(11), 613-620.
- Salton, G. and Buckley, C. (1987). Term Weighting Approaches in Automatic Text Retrieval. Technical Report 87-881. Ithaca, NY: Department of Computer Science, Cornell University.
- Salton, G., and Buckley, C. (1988). Parallel Text Search Methods. *Communications of the ACM*, 31(2), 202-215.
- Salton, G. (1989). *Automatic Text Processing*. Reading, MA: Addison-Wesley Publishing Company.
- Stanfill, C. and Kahle, B. (1986). Parallel Free-Text Search on the Connection Machine System. *Communications of the ACM*, 29(12), 1229-1239.
- Stanfill, C. (1988). Parallel Computing for Information Retrieval: Recent Developments. Technical Report DR88-1. Cambridge MA: Thinking Machines Corporation.
- Stanfill, C., R. Thau, and D. Waltz. (1989, June). A Parallel Indexed Algorithm for Information Retrieval. Paper presented at the International Conference on Research and Development in Information Retrieval. Boston, MA.
- Stanfill, C. (1990a). Information Retrieval Using Parallel Signature Files. *IEEE Data Engineering Bulletin*, 13(1), 33-40.
- Stanfill, C. (1990b, September). Partitioned Posting Files: A Parallel Inverted File Structure for Information Retrieval. Paper presented at the International Conference on Research and Development in Information Retrieval. Brussels, Belgium.
- Stone, H. (1987). Parallel Querying of Large Databases: A Case Study. *Computer*, 20(10), 11-21.
- Thinking Machines Corporation. (1987). Connection Machine Model CM-2 Technical Specifications. Cambridge MA: Thinking Machines Corporation.
- van Rijsbergen, C. J. (1979). *Information Retrieval*, 2nd ed. London: Butterworths.